# 编译原理实验三　实验报告

组长：白家杨 161220002 组员：段高磊 161220036

## 一、编译说明

本次实验不改变助教的Makefile文件，编译方法为直接在Code文件夹下执行make指令。

想要检测特定的程序文件，比如example.cmm，步骤为：先执行make编译（生成名为 **parser** 的可执行文件），然后执行以下命令（假定example.cmm文件与parser在同一目录下）：

```
./parse example.cmm 输出文件路径名(以“.ir”为后缀)
```

## 二、功能实现

1. 必做功能：基于C--语法的7个假设，对输入的文件进行词法分析、语法分析和语义分析的基础上，将C--代码翻译为中间代码

（1）以线性结构存储中间代码　（2）优化中间代码　（3）输出中间代码到文件（"*.ir"）

2. 选作**要求3.2**：修改C--语言假设3、假设2，使代码中可以出现一维数组作为函数参数，并且变量中可以出现高维数组。

## 三、实现过程

### 1. 中间代码的表示（线性）

本次实验我们选择线性的双向链表来记录中间代码。一条中间代码的数据结构定义为InterCode，成员kind表示分成19个类型，不同类型表示不同的中间代码表现形式，每种形式对应着成员u的赋值。u由操作数填充而成。类型和操作数对应具体表现形式可以参考intermediateCode.h文件中**printInterCodes（FILE* fw）**函数的实现。

对于操作数用结构体Operand_来定义，记录了这个操作数的类型以及该操作数在中间代码的表示。

双向链表中每一项就是一个InterCodes结构体，包含一行代码InterCode以及指向前一个和指向后一个InterCodes的指针。

```
typedef struct Operand_
{   enum { VAR, CONST, ADDR, TEMPVAR, LABEL, FUNC, AddrVarNo, PointVar, AddrParam,//表示地址的传参
            AddrVarName//表示需要加上*，变量名
    } kind;
    union { int no;
            char *val;//func, const, var
    } u;
} Operand_;
```

```
typedef struct InterCode
{   enum { DEC, //0
           CLABEL, FUNCTION, GOTO, RETURN, ARG, PARAM, READ, WRITE,
           ASSIGN, CALL,//9, 10
           ADD, SUB, MUL, DIV, //11, 12, 13, 14
           IFGOTO,//15
           getaddr, getpointer,//16, 17,represent result=&op1, result=*op1
           pointto//18, represent *result=op1
    } kind;
    union {
        struct { Operand op; int size; } dec;
        struct { Operand op; } single;
        struct { Operand left, right; } assign;
        struct { Operand result, op1, op2; } binop;
        struct { Operand op1,op2,label;
                 char relop[3];     } triop;
    } u;
} InterCode;
typedef struct InterCodes
{   InterCode code;
    struct InterCodes *prev, *next;
} InterCodes;
```

**2、翻译模式**

基于实验指导书，完成了基本表达式、语句、条件表达式、函数调用以及数组的翻译模式。

**3、选做任务的实现**

本次实现了多维数组的翻译。多维数组在词法分析中会分析出属于数组类型，并可以通过词法分析来计算数组需要的空间大小（计算函数**computeSize(Type t)** ）因而对于变量中出现的多维数组，可以通过在分析到词法VarDec时可以判断并进行开辟空间的DEC 操作。

| **translate_VarDec()** | |
|---|---|
| VarDec->ID | if （ID 不是一个函数参数且对应的是一个数组)<br>{ return：Dec ID computeSize(ID.type）} |

对于变量中数组的访问，由于词法分析中的递归访问，可以通过上一层返回的操作数类型是否是一个表示地址的操作数来判断是否需要进行取址操作。

| translate_Exp() | |
|---|---|
| Exp->Exp1[Exp2] | if( Exp1获得的是表示地址的变量 )<br>{ Exp.type = 地址;<br>return : Exp := Exp1 + Exp2 * ComputerSize(Exp1.type) }<br>if( Exp1获得的是表示变量 )<br>{ Exp.type = 地址;<br>return : Exp := &Exp1 + Exp2 * ComputerSize(Exp1.type); } |
| Exp->Exp1 ASSIGN Exp2 | if( Exp1.type == 地址 && Exp2.type== 地址 )<br>{ return *Exp1 := *Exp2 }<br>if( Exp1.type == 地址 && Exp2.type!= 地址 )<br>{ return *Exp1 := Exp2 }<br>if( Exp1.type != 地址 && Exp2.type== 地址 )<br>{ return Exp1 := *Exp2 }<br>else return Exp1:=Exp2 |
| Exp->Exp1 PLUS Exp2<br>（ Exp->Exp1 MINUS Exp2<br>Exp->Exp1 PLUS Exp2<br>Exp->Exp1 DIV Exp2 ） | if( Exp1.type == 地址 && Exp2.type== 地址 )<br>{ return Exp := *Exp1+ *Exp2 }<br>if( Exp1.type == 地址 && Exp2.type!= 地址 )<br>{ return Exp :=*Exp1 + Exp2 }<br>if( Exp1.type != 地址 && Exp2.type== 地址 )<br>{ return Exp := Exp1 + *Exp2 }<br>else return Exp:=Exp1+Exp2 |

**4.优化部分**

（1）优化数组地址的计算

对于数组中比如x[0][1]的情况,对应于语法分析中**Exp->Exp1[Exp2]** 的情况，可以将Exp2进行整合,减少譬如需要分别计算0和1相对于地址偏移值的代码量

（2）对于折叠的常量，直接计算结果并存储，减少中间临时变量语句的生成。

（3） 减少逻辑上冗余的赋值语句与算术表达式（后续未使用），如下述中删除第一条赋值语句：

```
y = 3;
... //中间不涉及变量y的使用
y = 10;
```